User's Manual for `elegant`

Program Version 12.4
Manual Version 1

—

Advanced Photon Source

—

Michael Borland—6 May 1993

1 Introduction

`elegant` stands for "ELEctron Generation ANd Tracking," a somewhat out-of-date description of a fully 6D accelerator program that now does much more than generate particle distributions and track them. `elegant`, written entirely in the C programming language[1], uses a variant of the MAD[2] input format to describe accelerators, which may be either transport lines, circular machines, or a combination thereof. Program execution is driven by commands in a namelist format.

This document describes the features available in `elegant`, listing the commands and their arguments. The differences between `elegant` and MAD formats for describing accelerators are listed. A series of examples of `elegant` input and output are given. Finally, appendices are included describing the post-processing programs.

1.1 Program Philosophy

For all its complexity, `elegant` is not a stand-alone program. For example, most of the output is not human-readable, and `elegant` itself has no graphics capabilities. These tasks are handled by a suite of post-processing programs that serve both `elegant` and other physics programs. The program `awe` (Access With Ease) is a data-extraction and manipulation program using a self-describing data format. `awe` will allow one to make customized tables containing data computed by `elegant`. This is, however, the least used of its capabilities. `awe` permits the user to define new data elements in terms of those present in a data file, and to process the data to extract overall properties (e.g., averages). Rather than printing the data in endless tables, `awe` is primarily used to put user-selected data pairs into a format acceptable by the general-purpose graphics program `mpl` (Multi-PLot).

`mpl` is itself not a stand-alone program, in that it is one of a group of about 30 programs sharing the `mpl` data format. These programs, collectively called the `mpl` Scientific Toolkit, allow manipulation of data beyond what is possible with `awe`. For example, several of the programs from the Toolkit permit one to perform arithmetic or higher operations using data files as terms in an equation. Other commonly used programs generate histograms and do statistical analysis.

Setting up for an `elegant` run thus involves more than creating input files for `elegant` per se. A complicated run will typically involve creation of a post-processing command file that processes `elegant` output and puts it in the most useful form, typically a series of graphs. Users thus have the full power of `awe`, the Toolkit, and the resident command interpreter (e.g., the UNIX shell) at their disposal. The idea is that instead of continually rewriting the physics code to, for example, make another type of graph or squeeze another item into a crowded table, one should allow the user to tailor the output to his specific needs using a set of generic post-processing programs. This approach has been quite successful, and is believed particularly suited to the constantly changing needs of research.

1.2 Capabilities of elegant

elegant started as a tracking code, and it is still well-suited to this task. elegant tracks in the 6-dimensional phase space $(x, x', y, y', s, \delta)$, where x (y) is the horizontal (vertical) transverse coordinate, primed quantities are slopes, s is the *total* distance traveled, and δ is the fractional momentum deviation[3]. Note that these quantities are commonly referred to as (x, xp, y, yp, s, dp) in the namelists, accelerator element parameters, and output files. ("dp" is admittedly confusing—it is supposed to remind the user of $\Delta P/P_o$.)

Tracking may be performed using matrices (of selectable order), canonical kick elements, numerically integrated elements, or any combination thereof. For most elements, second-order matrices are available; matrix concatenation can be done to any order up to third. Canonical kick elements are available for bending magnets, quadrupoles, sextupoles, and higher-order multipoles; all of these elements also support optional classical synchrotron radiation losses. Among the numerically integrated elements available are extended-fringe-field bending magnets and traveling-wave accelerators. A number of hybrid elements exist that have first-order transport with exact time dependence, e.g., RF cavities. Some of the more unusual elements available are third-order alpha-magnets[4, 5], time-dependent kicker magnets, voltage-ramped RF cavities, beam scrapers, and beam-analysis "screens."

A wide variety of output is available from tracking, including centroid and sigma-matrix output along the accelerator. In addition to tracking internally generated particle distributions, elegant can track distributions stored in external files, which can either be generated by other programs or by previous elegant runs. Because elegant uses awe format for reading in and writing out particle coordinates, it is relatively easy to interface elegant to other programs using files that can also be used with awe to do post-processing for the programs. Among the programs that have been interfaced to elegant in this fashion are rfgun, spiffe, MASK, PARMELA, and EGS4; none of this involved any modification of elegant itself.

elegant allows the addition of random errors to virtually any parameter of any accelerator element. One can correct the orbit (or trajectory), tunes, and chromaticity after adding errors, then compute Twiss parameters, track, or perform a number of other operations. Unlike many other programs, elegant allows one to make a single run simulating an arbitrary number of randomizations of an accelerator. Analysis of the resulting data is made relatively painless by use of the awe format for most output. Hence, instead of doing a few simulations with a few seed numbers, the user can do hundreds or even thousands of randomizations of one accelerator to get an accurate representation of the statistics, with no more work invested than in doing a few simulations[6].

In addition to randomly perturbing accelerator elements, elegant allows one to systematically vary any number of elements in a multi-dimensional grid. As before, one can track or do other computations for each point on the grid. This is a very useful feature for the simulation of experiments, e.g., emittance measurements involving beam-size measurements during variation of one or more quadrupoles[6].

Like many accelerator codes, elegant does accelerator optimization. While elegant is not designed to replace first-order matching codes like MAD, it will fit the first-order matrix. Of more interest is the ability to optimize results of tracking using a user-supplied function of the final beam and transport parameters. This permits solution of a wide variety of problems, from matching a kicker bump in the presence of nonlinearities to optimizing dynamic aperture by adjusting sextupoles.

elegant provides several methods for determining accelerator aperture, whether dynamic or physical. One may do straightforward tracking of an ensemble of particles that occupies at uniform grid in (x, y) space. A more efficient variant of this procedure involves tracking a series of constantx lines of particles with fixed y values, with elimination of any given y value whenever a stable particle is found. Finally, one may use a single-particle search method that can locate the aperture for a series of y values, to a predefined resolution in x.

In addition to using analytical expressions for the transport matrices, `elegant` supports computation of the first-order matrix and linear optics properties of a circular machine based on tracking. A common application of this is to compute the tune and beta-function variation with momentum offset by single-turn tracking of a series of particles. This is much more efficient than, for example, tracking and performing FFTs (though `elegant` will do this also). This both tests analytical expressions for the chromaticity and allows computations using accelerator elements for which such expressions do not exist (e.g., a numerically integrated bending magnet with extended fringe fields).

A common application of random error simulations is to set tolerances on magnet strength and alignment relative to the correctability of the closed orbit. A more efficient way to do these calculations is to use correct-orbit amplification factors[6]. `elegant` the computes amplification factors and functions for corrected and uncorrected orbits and trajectories pertaining to any element that produces an orbit or trajectory distortion. It simultaneously computes the amplification functions for the steering magnets, in order to determine how strong the steering magnets will need to be.

2 Namelist Command Dictionary

The main input file for an `elegant` run consists of a series of namelists, which function as commands. Most of the namelists direct `elegant` to set up to run in a certain way. A few are "action" commands that begin the actual simulation. FORTRAN programmers should note that, unlike FORTRAN namelists, these namelists need not come in a predefined order; `elegant` is able to detect which namelist is next in the file and process appropriately.

Each namelist has a number of variables associated with it, which are used to control details of the run. These variables come in three data types: (1) `long`, for the C long integer type, (2) `double`, for the C double-precision floating point type, and (3) `STRING`, for a character string enclosed in double quotation marks. All variables have default values, which are listed on the following pages. `STRING` variables often have a default value listed as `NULL`, which means no data; this is quite different from the value "", which is a zero-length character string. `long` variables are often used as logical flags, with a zero value indicating false and a non-zero value indicating true.

On the following pages the reader will find individual descriptions of each of the namelist commands and their variables. Each description contains a sequence of the form

```
&<namelist-name>
    <variable-type> <variable-name> = <default-value>;
    .
    .
    .
&end
```

This summarizes the parameters of the namelist. Note, however, that the namelists are invoked in the form

```
&<namelist-name>
    [<variable-name> = <value> ,]
    [<array-name>[<index>] = <value> [,<value> ...] ,]
        .
```

```
          .
          .
&end
```

The square-brackets enclose an optional component. Not all namelists require variables to be given–the defaults may be sufficient. However, if a variable name is given, it must have a value. Values for `STRING` variables must be enclosed in double quotation marks. Values for `double` variables may be in floating-point, exponential, or integer format (exponential format uses the 'e' character to introduce the exponent).

Array variables take a list of values, with the first value being placed in the slot indicated by the subscript. As in C, the first slot of the array has subscript 0, *not* 1. The namelist processor does not check to ensure that you do not put elements into nonexistent slots beyond the end of the array; doing so may cause the processor to hang up or crash.

Wildcards are allowed in a number of places in `elegant`, `awe`, and the `mpl` Toolkit. The wildcard format is very similar to that used in UNIX:

- `*` — stands for any number of characters, including none.
- `?` — stands for any single character.
- `[<list-of-characters>]` — stands for any single character from the list. The list may include ranges, such as `a-z`, which includes all characters between and including 'a' and 'z' in the ASCII character table.

The special characters `*`, `?`, `[`, and `]` are entered literally by preceeding the character by a backslash (e.g., `\*`).

In many places where a filename is required in an `elegant` namelist, the user may supply a so-called "incomplete" filename. An incomplete filename has the sequence "%s" imbedded in it, for which is substituted the "rootname." The rootname is by default the filename (less the extension) of the lattice file. The most common use of this feature is to cause `elegant` to create names for all output files that share a common filename but differ in their extensions. Post-processing can be greatly simplified by adopting this naming convention, particularly if one consistently uses the same extension for the same type of output. Recommended filename extensions are given in the lists below.

When `elegant` reads a namelist command, one of its first actions is to print the namelist back to the standard output. This printout includes all the variables in the namelist and their values. Occasionally, the user may see a variable listed in the printout that is not in this manual. These are often obsolete and are retained only for backward compatibility, or else associated with a feature that is not fully supported. Use of such "undocumented features" is discouraged.

amplification_factors

- type: action command.
- function: compute corrected and uncorrected orbit amplification factors and functions.

```
&amplification_factors
    STRING output = NULL;
    STRING uncorrected_orbit_function = NULL;
    STRING corrected_orbit_function = NULL;
    STRING kick_function = NULL;
    STRING name = NULL;
    STRING type = NULL;
    STRING item = NULL;
    STRING plane = NULL;
    double change = 1e-3;
    long number_to_do = -1;
    double maximum_z = 0;
&end
```

- `output` — The (incomplete) name of a file for text output. Recommended value: "%s.af".
- `uncorrected_orbit_function` — The (incomplete) name of a file for an `mpl`-format output of the uncorrected-orbit amplification function. Recommended value: "%s.uof".
- `corrected_orbit_function` — The (incomplete) name of a file for an `mpl`-format output of the corrected-orbit amplification function. Recommended value: "%s.cof".
- `kick_function` — The (incomplete) name of a file for an `mpl`-format output of the kick amplification function. Recommended value: "%s.kaf".
- `name` — The optionally wildcarded name of the orbit-perturbing elements.
- `type` — The optional type name of the the orbit-perturbing elements.
- `item` — The parameter of the elements producing the orbit.
- `plane` — The plane ("h" or "v") to examine.
- `change` — The parameter change to use in computing the amplification.
- `number_to_do` — The number of elements to perturb.
- `maximum_z` — The maximum z coordinate of the elements to perturb.

analyze_map

- type: action command.
- function: find the approximate first-order matrix and related quantities for an accelerator by tracking.

```
&analyze_map
    STRING output = NULL;
    double delta_x = 1e-6;
    double delta_xp = 1e-6;
    double delta_y = 1e-6;
    double delta_yp = 1e-6;
    double delta_s  = 1e-6;
    double delta_dp = 1e-6;
    long center_on_orbit = 0;
    long verbosity = 0;
&end
```

- `output` — The (incomplete) name of a file for `awe`-format output.
  - Recommended value: "%s.ana".
  - File contents: A series of dumps, each consisting of a single data point containing the centroid offsets for a single turn, the single-turn R matrix, the matched Twiss parameters, tunes, and dispersion functions.
- `delta_X` — The amount by which to change the quantity X in computing the derivatives that give the matrix elements.
- `center_on_orbit` — A flag directing the expansion to be made about the closed orbit instead of the design orbit.
- `verbosity` — The larger this value, the more output is printed during computations.

awe_beam

- type: setup command.
- function: set up for tracking of particle coordinates stored in an awe-format file.

```
&awe_beam
    STRING input = NULL;
    STRING input_type = "elegant";
    long n_particles_per_ring = 1;
    long one_random_bunch = 0;
    long prebunched = 0;
    long sample_interval = 1;
    long n_dumps_to_skip = 0;
    long center_transversely = 0;
    double sample_fraction = 1;
    double p_lower = 0.0;
    double p_upper = 0.0;
    long use_multipliers = 0;
    double x_multiplier = 1.0;
    double xp_multiplier = 1.0;
    double y_multiplier = 1.0;
    double yp_multiplier = 1.0;
    double dp_multiplier = 1.0;
&end
```

- `input` — Name of an awe-format file containing coordinates of input particles.
- `input_type` — May be "elegant", "spiffe", or "mask", indicating the name of the program that wrote the input file. The expected data quantities for the different types are:
  - `elegant`: (x, xp, y, yp, t, p), where x and y are in meters, $xp = x'$ and $xp = y'$ are dimensionless, t is in picoseconds, and $p = \beta\gamma$ is the dimensionless momentum.
  - `spiffe`: (r, z, pr, pz, t), where r and z are in meters, $pr = \beta_r\gamma$, $pz = \beta_z\gamma$, and t is in picoseconds.
  - MASK: (x1, x2, beta1, beta2, t, p), where x1 and x2 are in meters, t is in picoseconds, and $p = \beta\gamma$. Direction 1 is longitudinal (z) and direction 2 is transverse (r).
- `n_particles_per_ring` — For `spiffe` or `mask` data, gives the number of particles to generate for each ring of charge.
- `one_random_bunch` — A flag indicating whether, for `spiffe` or `mask` data, a new random distribution should be calculated for each step of the simulation.
- `prebunched` — A flag indicating, if zero, that the entire file is one "bunch," and otherwise that each `awe` dump in the file is a different bunch.
- `sample_interval` — If non-zero, only every `sample_interval`th particle is used.
- `n_dumps_to_skip` — Number of `awe` dumps to skip at the beginning of the file.

- `center_transversely` — If non-zero, the transverse centroids of the distribution are made to be zero.
- `sample_fraction` — If non-unity, the randomly selected fraction of the distribution to use.
- `p_lower`, `p_upper` — If different, the lower and upper limit on $\beta\gamma$ of particles to use.
- `use_multipliers` — If non-zero, the phase-space coordinates of each particle are multiplied by the factors `x_multiplier`, `xp_multiplier`, `y_multiplier`, `yp_multiplier`, and `dp_multiplier`, as appropriate.

bunched_beam

- type: setup command.
- function: set up for tracking of particle coordinates with various distributions.

```
&bunched_beam
    STRING bunch = NULL;
    long n_particles_per_bunch = 1;
    double time_start = 0;
    STRING matched_to_cell = NULL;
    double emit_x  = 0;
    double beta_x  = 1.0;
    double alpha_x = 0.0;
    double eta_x   = 0.0;
    double etap_x  = 0.0;
    double emit_y  = 0;
    double beta_y  = 1.0;
    double alpha_y = 0.0;
    double eta_y   = 0.0;
    double etap_y  = 0.0;
    double Po = 5.0;
    double sigma_dp = 0.0;
    double sigma_s = 0.0;
    double dp_s_coupling = 0;
    long one_random_bunch = 1;
    long limit_invariants = 0;
    long symmetrize = 0;
    long enforce_rms_values[3] = {0, 0, 0};
    double distribution_cutoff[3] = {2, 2, 2};
    STRING distribution_type[3] = {"gaussian","gaussian","gaussian"};
    double centroid[6] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
&end
```

- `bunch` — The (incomplete) name of an awe-format file to which the phase-space coordinates of the bunches are to be written.
  - Recommended value: "%s.bun".
  - File contents: A series of dumps (one for each bunch generated), each with `n_particles` data points consisting of (x, x', y, y', t, p) for the particles.
- `n_particles_per_bunch` — Number of particles in each bunch.
- `time_start` — The central value of the time coordinate for the bunch.
- `matched_to_cell` — The name of a beamline from which the Twiss parameters of the bunch are to be computed.
- `emit_X` — RMS emittance for the X plane.
- `beta_X`, `alpha_X`, `eta_X`, `etap_X` — Twiss parameters for the X plane.

- Po — Central momentum of the bunch.
- sigma_dp, sigma_s — Fractional momentum spread, δ, and bunch length.
- dp_s_coupling — Specifies the coupling between s and δ, defined as $\langle s\delta\rangle/(\sigma_s\sigma_\delta)$.
- one_random_bunch — If non-zero, then only one random particle distribution is generated. Otherwise, a new distribution will be generated for every simulation step.
- enforce_rms_values[3] — Flags, one for each plane, indicating whether to force the distribution to have the specified RMS properties.
- distribution_cutoff[3] — Distribution cutoff parameters for each plane.
- distribution_type[3] — Distribution type for each plane. May be "gaussian", "hard-edge", "uniform-ellipse", "shell", or "dynamic-aperture".
- limit_invariants — If non-zero, the distribution cutoffs are applied to the invariants, rather than to the coordinates.
- symmetrize — If non-zero, the distribution is symmetric under changes of sign in the coordinates.
- centroid[6] — Centroid offsets for each of the six coordinates.

chromaticity

- type: setup command.
- function: set up for chromaticity correction.

```
&chromaticity
    STRING sextupoles = NULL;
    double dnux_dp = 0;
    double dnuy_dp = 0;
    double sextupole_tweek = 1e-3;
    long n_iterations = 1;
    STRING strength_log = NULL;
&end
```

- `sextupoles` — List of names of elements to use to correct the chromaticities.
- `dnux_dp`, `dnuy_dp` — Desired chromaticity values.
- `sextupole_tweek` — Amount by which to tweak the sextupoles to compute derivatives of chromaticities with respect to sextupole strength. [The word "tweak" is misspelled "tweek" in the code.]
- `n_iterations` — Number of iterations of the correction to perform.
- `strength_log` — The (incomplete) name of a column format file to which the sextupole strengths will be written. Recommended value: "%s.sst".

closed_orbit

- type: setup command.
- function: set up for computation of the closed orbit.

```
&closed_orbit
    STRING output = NULL;
    long start_from_centroid = 1;
    double closed_orbit_accuracy = 1e-12;
    long closed_orbit_iterations = 10;
    long fixed_length = 0;
    long start_from_recirc = 0;
    long verbosity = 0;
&end
```

- `output` — The (incomplete) name of an `awe`-format file to which the closed orbits will be written.
  - Recommended value: "%s.clo".
  - File contents: A series of dumps (one for each simulation step), each containing the horizontal and vertical closed orbit in the form of a series of (z,x,y) points. The data point label for each point is the name of the element which ends at the given z value.
- `start_from_centroid` — A flag indicating whether to force the computation to use the momentum centroid of the beam distribution.
- `closed_orbit_accuracy` — The desired accuracy of the closed orbit.
- `closed_orbit_iterations` — The number of iterations to take in finding the closed orbit.
- `fixed_length` — A flag indicating whether to find a closed orbit with the same length as the design orbit by changing the momentum offset.
- `start_from_recirc` — A flag indicating whether to compute the closed orbit from the `recirc` element in the beamline.
- `verbosity` — A larger value results in more output during the computations.

correct

- type: setup command.
- function: set up for correction of the trajectory or closed orbit.

```
&correct
    STRING mode = "trajectory";
    STRING method = "global";
    STRING trajectory_output = NULL;
    STRING corrector_output = {NULL};
    STRING statistics[2] = {NULL, NULL};
    double corrector_tweek[2] = {1e-3, 1e-3};
    double corrector_limit[2] = {0, 0};
    double correction_fraction[2] = {1, 1};
    double correction_accuracy[2] = {1e-6, 1e-6};
    double bpm_noise[2] = {0, 0};
    double bpm_noise_cutoff[2] = {1.0, 1.0};
    STRING bpm_noise_distribution[2] = {"uniform", "uniform"};
    long verbose = 1;
    long fixed_length = 0;
    long n_xy_cycles = 1;
    long n_iterations = 1;
    long prezero_correctors = 1;
    long track_before_and_after = 0;
    long start_from_centroid = 1;
    long use_actual_beam = 0;
    double closed_orbit_accuracy = 1e-12;
    long closed_orbit_iterations = 10;
&end
```

- `mode` — Either "trajectory" or "orbit", indicating correction of a trajectory or a closed orbit.
- `method` — For trajectories, may be "one-to-one" or "global". For closed orbit, must be "global".
- `trajectory_output` — The (incomplete) name of an awe-format file to which the trajectories or orbits will be written.
  - Recommended value: "%s.trj" or "%s.orb".
  - File contents: A series of dumps (two for each simulation step), each containing the orbit or trajectory as a series of points of the form (z, x, y, n), where n is the number of particles (for correction by tracking a distribution). The dump label is either "corrected" or "uncorrected". The data point label is the name of the element that ends at the given z location.
- `corrector_output` — The (incomplete) name of an awe-format file to which information about the final corrector strengths will be written.
  - Recommended value: "%s.cor".

  - File contents: A series of dumps (several for each simulation step), each containing the steering element strengths as a series of points of the form (z, kick). The dump label is either "horizontal" or "vertical". The data point label is of the form `<corrector_name>[occurrence][<parameter_name>]`, giving the name and occurrence of the steering element ending at given z location.

- `statistics[2]` — The (incomplete) name of an awe-format file to which statistical information about the trajectories (or orbits) and corrector strengths will be written.
  - Recommended value: "%s.xcor" and "%s.ycor".
  - File contents: A series of dumps (one for each correction cycle), each containing statistical information on the correction in the form a series of data points containing (iteration, cycle, RMS kick, RMS position, maximum kick, maximum position, momentum offset). The dump label is either "intermediate cycle" or "final cycle". The data point label is one of "uncorrected", "corrected", and "intermediate".
- `corrector_tweek[2]` — The amount by which to change the correctors in order to compute correction coefficients. [The word "tweak" is misspelled "tweek" in the code.]
- `corrector_limit[2]` — The maximum strength allowed for a corrector.
- `correction_fraction[2]` — The fraction of the computed correction strength to actually use for any one iteration.
- `correction_accuracy[2]` — The accuracy of the correction in terms of the RMS BPM values.
- `bpm_noise[2]` — The BPM noise level.
- `bpm_noise_cutoff[2]` — Cutoff values for the random distributions of BPM noise.
- `bpm_noise_distribution[2]` — May be either "gaussian", "uniform", or "plus_or_minus".
- `verbose` — If non-zero, information about the correction is printed during computations.
- `fixed_length` — Indicates that the closed orbit length should be kept the same as the design orbit length by changing the momentum offset of the beam.
- `n_xy_cycles` — Number of times to alternate between correcting the x and y planes.
- `n_iterations` — Number of iterations of the correction.
- `prezero_correctors` — Flag indicating whether to set the correctors to zero before starting.
- `track_before_and_after` — Flag indicating whether tracking will be done both before and after correction.
- `start_from_centroid` — Flag indicating that correction should start from the beam centroid. For orbit correction, only the beam momentum centroid is relevant.
- `use_actual_beam` — Flag indicating that correction should employ tracking of the beam distribution rather than a single particle.
- `closed_orbit_accuracy` — Accuracy of closed orbit computation.
- `closed_orbit_iterations` — Number of iterations of closed orbit computation.

correct_tunes

- type: setup command.
- function: set up for correction of the tunes.

```
&correct_tunes
    STRING quadrupoles = NULL;
    double tune_x = 0;
    double tune_y = 0;
    long n_iterations = 1;
    STRING strength_log = NULL;
&end
```

- `quadrupoles` — List of names of quadrupoles to be used.
- `tune_x`, `tune_y` — Desired x and y tune values. If not given, the desired values are assumed to be the unperturbed tunes.
- `n_iterations` — The number of iterations of the correction to perform.
- `strength_log` — The (incomplete) name of a `column`-format file to which the quadrupole strengths will be written as correction proceeds. Recommended value: "%s.qst".

error

- type: setup command.
- function: assert a random error defintion for the accelerator.

```
&error
    STRING name = NULL;
    STRING item = NULL;
    STRING type = "gaussian";
    double amplitude = 0.0;
    double cutoff = 3.0;
    long bind = 1;
    long bind_number = 0;
    long fractional = 0;
    long post_correction = 0;
&end
```

- `name` — The possibly wildcarded name of the elements for which errors are being specified.
- `item` — The parameter of the elements to which the error pertains.
- `type` — The type of random distribution to use. May be one of "uniform", "gaussian", or "plus_or_minus". A "plus_or_minus" error is equal in magnitude to the amplitude given, with the sign randomly chosen.
- `amplitude` — The amplitude of the errors.
- `cutoff` — The cutoff for the random distribution.
- `bind`, `bind_number` — If `bind_number` is positive, then a positive value of `bind` indicates that `bind_number` successive elements having the same name will have the same error value. If `bind` is negative, then the sign of the error is alternated between successive elements.
- `fractional` — A flag indicating whether the errors are fractional, in which case the amplitude refers to the amplitude of the fractional error.
- `post_correction` — A flag indicating whether the errors should be added after orbit, tune, and chromaticity correction.

error_control

- type: setup command
- function: overall control of random errors.

```
&error_control
    long clear_error_settings = 1;
    long summarize_error_settings = 0;
    STRING error_log = NULL;
&end
```

- `clear_error_settings` — Clear all previous error settings.
- `summarize_error_settings` — Summarize current error settings.
- `error_log` — The (incomplete) name of a `column`-format file to which error values will be written. Recommended value: "%s.erl".

find_aperture

- type: action command.
- function: find the aperture in (x, y) space for an accelerator.

```
&find_aperture
    STRING output = NULL;
    STRING boundary = NULL;
    STRING mode = "many-particle";
    double xmin = -0.1;
    double xmax =  0.1;
    double ymin =  0.0;
    double ymax =  0.1;
    long nx  = 21;
    long ny  = 11;
    long n_splits = 1;
    double split_fraction = 0.5;
    double desired_resolution = 0.01;
    long verbosity = 0;
    long assume_nonincreasing = 0;
&end
```

- `output` — The (incomplete) name of an `awe`-format file to send output to.
  - Recommended value: "%s.aper".
  - File contents: A series of dumps (one for each simulation step), each containing the (x, y) coordinates of a series of stable or accepted points on the aperture boundary.
- `boundary` — The (incomplete) name of an `mpl`-format file for the boundary points of the aperture search. Recommended value: "%s.bnd".
- `xmin`, `xmax`, `ymin`, `ymax` — Region of the aperture search.
- `mode` — May be "many-particle" or "single-particle". Many-particle searching is much faster, but does not allow interval splitting to search for the aperture boundary.
- `nx` — Number of x values to take in initial search.
- `ny` — Number of y values to take in search.
- `n_splits` — If positive, the number of times to do interval splitting.
- `split_fraction` — If interval splitting is done, how the interval is split.
- `desired_resolution` — If interval splitting is done, fraction of `xmax-xmin` to which to resolve the aperture.
- `assume_nonincreasing` — If interval splitting is done and if this variable is non-zero, the search assumes that the aperture at $y + \Delta y$ is no larger than that at y.
- `verbosity` — A larger value results in more output during computations.

link_control

- type: setup command.
- function: overall control of element parameter links.

```
&link_control
    long clear_links = 1;
    long summarize_links = 0;
    long verbosity = 0;
&end
```

- `clear_links` — Clear all previously set links.
- `summarize_links` — Summarize all current set links.
- `verbosity` — A larger value results in more output during computations.

link_elements

- type: setup command.
- function: assert a link between parameters of accelerator elements.

```
&link_elements
    STRING target = NULL;
    STRING item = NULL;
    STRING source = NULL;
    STRING source_position = "before";
    STRING mode = "dynamic";
    STRING equation = NULL;
&end
```

- `target` — The name of the elements to be modified by the link.
- `item` — The parameter that will be modified.
- `source` — The name of the elements to be linked to.
- `source_position` — May be one of "before", "after", "adjacent", "nearest", or "same-occurrence".
- `mode` — May be either "dynamic" or "static". A dynamic link is asserted whenever the source is changed (during correction, for example). A static link is asserted only when an error or variation is imparted to the source, and at the end of correction.
- `equation` — A `rpn` equation for the item value in terms of the item values for the source. To refer to the source parameter values, use the form `<source-name>[<item-name>]`; these sequences must appear in capital letters.

matrix_output

- type: setup/action command.
- function: generate matrix output, or set up to do so later.

```
&matrix_output
    STRING printout = NULL;
    long printout_order = 1;
    long full_matrix_only = 0;
    STRING awe_output = NULL;
    long awe_output_order = 1;
    long output_at_each_step = 0;
    STRING start_from = NULL;
    long start_from_occurrence = 1;
&end
```

- `printout` — The (incomplete) name of a file to which the matrix output will be printed (as text). Recommended value: "%s.mpr".
- `printout_order` — The order to which the matrix is printed.
- `full_matrix_only` — A flag indicating that only the matrix of the entire accelerator is to be output.
- `awe_output` — The (incomplete) name of an `awe`-format file to which the matrix will be written.
  - Recommended value: "%s.mat".
  - File contents: A single dump, or (if `output_at_each_step` is nonzero) a series of dumps, each consisting of a series of data points containing `(z, C-matrix [, R-matrix [,T-matrix]])`, depending on the value of `awe_output_order`. C is the vector centroid offset at the end of the accelerator. The data point label is the name of the element that ends at the given z value.
- `awe_output_order` — The order to which the matrix is output in awe format.
- `output_at_each_step` — A flag indicating whether matrix output is desired at every simulation step.
- `start_from` — The optional name of the accelerator element from which to begin concatenation and output.
- `start_from_occurrence` — If `start_from` is not NULL, the number of the occurrence of the named element from which to start.

optimize

- type: action command.
- function: perform optimization.

```
&optimize
    long summarize_setup = 0;
&end
```

- summarize_setup — A flag indicating, if set, that a summary of the optimization parameters should be printed.

optimization_constraint

- type: setup command.
- function: define a constraint for optimization.

```
&optimization_constraint
    STRING quantity = NULL;
    double lower = 0;
    double upper = 0;
&end
```

- `quantity` — The quantity to be constrained, given as a `rpn` expression in terms of the optimization variables, the optimization covariables, and and the "final" parameters (see the entry for `run_setup` for the last of these). The optimization (co)variables are referred to as `<element-name>[<parameter-name>]`, in all capital letters.
- `lower`, `upper` — The lower and upper limits allowed for the expression.

optimization_covariable

- type: setup command.
- function: define an element parameter to be varied as a function of optimization parameters.

```
&optimization_covariable
    STRING name = NULL;
    STRING item = NULL;
    STRING equation = NULL;
&end
```

- `name` — The name of the element.
- `item` — The parameter of the element to be changed.
- `equation` — A `rpn` equation for the value of the parameter in terms of the values of any parameters of any optimization variable. These latter appear in the equation in the form `<element-name>[<parameter-name>]`, in all capital letters.

optimization_setup

- type: setup command.
- function: define overall optimization parameters and methods.

```
&optimization_setup
    STRING equation = NULL;
    STRING mode = "minimize";
    STRING method = "simplex";
    double tolerance = -0.01;
    double target = 0;
    long soft_failure = 1;
    long n_passes = 2;
    long n_evaluations = 500;
    STRING log_file = NULL;
&end
```

- `equation` — A rpn equation for the optimization function, expressed in terms of any parameters of any optimization variables and the "final" parameters of the beam (as recorded in the `final` output file available in the `run_setup` namelist). The optimization variables appear in the equation in the form `<element-name>[<parameter-name>]`.
- `mode` — May be either "minimize" or "maximize".
- `method` — May be one of "simplex", "grid", and "sample".
- `tolerance` — The convergence criterion for the optimization, with a negative value indicating a fractional criterion.
- `target` — The value which, if reached, results in immediate termination of the optimization, whether it has converged or not.
- `soft_failure` — A flag indicating, if set, that failure of an optimization pass should not result in termination of the optimization.
- `n_passes` — The number of optimization passes made to achieve convergence ("simplex" only).
- `n_evaluations` — The number of allowed evaluations of the optimization function. If simplex optimization is used, this is the number of allowed evaluations per pass.
- `log_file` — A file to which progress reports will be written as optimization proceeds.

optimization_variable

- type: setup command.
- function: defines a parameter of an element to be used in optimization.

```
&optimization_variable
    STRING name = NULL;
    STRING item = NULL;
    double lower_limit = 0;
    double upper_limit = 0;
    double step_size = 1;
&end
```

- `name` — The name of the element.
- `item` — The parameter of the element to be varied.
- `lower_limit`, `upper_limit` — The lower and upper limits allowed for the parameter. If these are equal, the range of the parameter is unlimited.
- `step_size` — The initial step size ("simplex" optimization) or the grid size in this dimension ("grid" or "sample" optimization).

`print_dictionary`

- type: action command.
- function: print dictionary of recognized accelerator elements.

```
&print_dictionary
    STRING filename = NULL;
&end
```

- `filename` — The name of a text file to which the dictionary will be printed.

rpn_expression

- type: action/setup command.
- function: pass an expression directly to the rpn submodule for execution.

```
&rpn_expression
    STRING expression = NULL;
&end
```

- `expression` — A `rpn` expression. This expression is executed immediately and can be used, for example, to read in `rpn` commands from a file or store values in `rpn` memories.

save_lattice

- type: action command.
- function: save the current accelerator element and beamline definitions.

```
&save_lattice
    STRING filename = NULL;
&end
```

- `filename` — The (incomplete) name of a file to which the element and beamline definitions will be written. Recommended value: "%s.new".

steering_element

- type: setup command.
- function: setup for use of a given parameter of a given element as a steering corrector.
- note: any use of this command disables the built-in definition of HKICK, VKICK, and HVKICK elements as steering elements.

```
&steering_element
    STRING name = NULL;
    STRING item = NULL;
    STRING plane = "h";
    double tweek = 1e-3;
    double limit = 0;
    STRING strength_log = NULL;
&end
```

- `name` — The name of the element.
- `item` — The parameter of the element to be varied.
- `plane` — May be either "h" or "v", for horizontal or vertical correction.
- `tweek` — The amount by which to change the item to compute the steering strength.
- `limit` — The maximum allowed absolute value of the item.
- `strength_log` — The (incomplete) name of a `column`-format file to which the strengths of the item will be written as correction proceeds.

trace

- type: setup command.
- function: set up for tracing of program execution.
- note: this option can dramatically slow down execution.

```
&trace
    long trace_on = 1;
    long heap_verify_depth = 0;
    STRING filename = NULL;
&end
```

- trace_on — A flag indicating, if set, that tracing should be activated.
- heap_verify_depth — The depth of subroutine calls to which memory heap checking should be performed.
- filename — The name of a file in which the call stack will be recorded.

twiss_output

- type: action/setup command.
- function: compute and output Twiss parameters, or set up to do so.

```
&twiss_output
    STRING filename = NULL;
    long matched = 1;
    long output_at_each_step = 0;
    long output_before_tune_correction = 0;
    long final_values_only = 0;
    double beta_x = 1;
    double alpha_x = 0;
    double eta_x = 0;
    double etap_x = 0;
    double beta_y = 1;
    double alpha_y = 0;
    double eta_y = 0;
    double etap_y = 0;
&end
```

- `filename` — The (incomplete) name of an `awe`-format file to which the Twiss parameters will be written.
  - Recommended value: "%s.twi".
  - File contents: A single dump, or (if `output_at_each_step` is nonzero) a series of dumps, each containing a series of data points giving a z location and the Twiss parameters at that location. The data point label is the name of the element that ends at the given z location. Each dump has auxiliary variables giving the tunes, chromaticities, and acceptances. If `final_values_only` is nonzero, only the values for the last z location are present in the tables.
- `matched` — A flag indicating, if set, that the periodic or matched Twiss parameters should be found.
- `output_at_each_step` — A flag indicating, if set, that output is desired at each step of the simulation.
- `output_before_tune_correction` — A flag indicating, if set, that output is desired both before and after tune correction.
- `final_values_only` — A flag indicating, if set, that only the final values of the Twiss parameters should be output, and not the parameters as a function of z.
- `beta_X`, `alpha_X`, `eta_X`, `etap_X` — If `matched` is zero, the initial values for the X plane.

run_control

- type: setup command.
- function: set up the number of simulation steps and passes.

```
&run_control
    long n_steps = 1;
    double bunch_frequency = 0;
    long n_indices = 0;
    long n_passes = 1;
&end
```

- `n_steps` — The number of separate repetitions of the action implied by the next action command. If random errors are defined, this is also the number of separate error ensembles.
- `bunch_frequency` — The frequency to use in calculating the time delay between repetitions.
- `n_indices` — The number of looping indices for which to expect definitions in subsequent `vary_element` commands.
- `n_passes` — The number of passes to make through the beamline per repetition.

run_setup

- type: setup command.
- function: set global parameters of the simulation and define primary input and output files.

```
&run_setup
    STRING lattice = NULL;
    STRING use_beamline = NULL;
    STRING rootname = NULL;
    STRING output = NULL;
    STRING centroid = NULL;
    STRING sigma = NULL;
    STRING final = NULL;
    STRING acceptance = NULL;
    STRING losses = NULL;
    STRING magnets = NULL;
    long combine_bunch_statistics = 0;
    long wrap_around = 1;
    long default_order = 2;
    long concat_order = 0;
    long print_statistics = 0;
    long random_number_seed = 987654321;
    long correction_iterations = 1;
    double p_central = 0.0;
    STRING expand_for = NULL;
&end
```

- `lattice` — Name of the lattice definition file.
- `use_beamline` — Name of the beamline to use.
- `rootname` — Filename fragment used in forming complete names from incomplete filenames. By default, the filename minus extension of the input file is used.
- `output` — The (incomplete) name of an awe-format file into which final phase-space coordinates will be written.
  - Recommended value: "%s.out".
  - File contents: A series of dumps (one for each simulation step), each containing a series of data points of the form (x, x', y, y', t, p). Each point corresponds to one particle that was transmitted through the entire run.
- `centroid` — The (incomplete) name of an awe-format file into which beam centroids as a function of z will be written.
  - Recommended value: "%s.cen".
  - File contents: A series of dumps (one for each simulation step), each consisting of a series of data points containing $(z, p_o, n, \langle x\rangle, \langle x'\rangle, \langle y\rangle, \langle y'\rangle, \langle s\rangle, \langle\delta\rangle)$. These are, respectively, the z location, the central momentum, the number of particles, the average position and slope

in the two planes, the average distance traveled, and the average momentum offset. The data point label for each point is the name of the element that ends at the given z location.

- `sigma` — The (incomplete) name of an `awe`-format file into which the beam sigma matrix as a function of z will be written.
  - Recommended value: "%s.sig".
  - File contents: A series of dumps (one for each simulation step), each consisting of a series of data points containing $(z, \mathbf{Sigma}, \mathbf{M}, \sigma)$. These are, respectively, the z location, the sigma matrix, the maximum particle amplitudes, and the RMS beam sizes. Note that the diagonal elements of the sigma matrix have had the square-root taken. The data point label for each point is the name of the element that ends at the given z location.

- `final` — The (incomplete) name of an `awe`-format file into which final beam and transport parameters will be written.
  - Recommended value: "%s.fin".
  - File contents: A series of dumps (one for each simulation step), each consisting of a single data point containing, in order, the RMS beam sizes, the beam centroids, the off-diagonal elements of the beam sigma matrix, the number of particles, the transmission fraction, the lattice (or central) momentum, the average beam momentum, the average beam kinetic energy, the geometric and normalized RMS emittances, the transverse beam widths, the bunch length, the total fractional bunch momentum interval, and the elements of the R matrix. The specific names of these so-called "final" parameters can be obtained using the `-list` option of `awe`. Note that the "final" parameters can be used in optimization expressions (see the entry for `optimization_setup`).

- `acceptance` — The (incomplete) name of an `awe`-format file into which the initial coordinates of transmitted particles will be written.
  - Recommended value: "%s.acc".
  - File contents: A series of dumps (one for each simulation step), each consisting of a series of points containing the initial phase-space coordinates of all accepted (i.e., transmitted) particles.

- `losses` — The (incomplete) name of an `awe`-format file into which information on lost particles will be written.
  - Recommended value: "%s.los".
  - File contents: A series of dumps (one for each simulation step), each consisting of a series of points containing the z location and the phase-space coordinates of a lost particle at the time of loss.

- `magnets` — The (incomplete) name of an `mpl`-format file into which a magnet layout representation will be written. Recommended value: "%s.mag".

- `combine_bunch_statistics` — A flag indicating whether to combine statistical information for all simulation steps. If non-zero, then the `sigma` and `centroid` data will be combined over all simulation steps.

- wrap_around — A flag indicating whether the z coordinate should wrap-around or increase monotonically in multipass simulations.
- default_order — The default order of transfer matrices used for elements having matrices.
- concat_order — If non-zero, the order of matrix concatenation used.
- print_statistics — A flag indicating whether to print information as each element is tracked.
- random_number_seed — A seed for the random number generators. If zero, a seed will be generated from the system clock.
- correction_iterations — Number of iterations in tune and chromaticity correction.
- p_central — Central momentum of the beamline, about which expansions are done.
- expand_for — Name of an awe-format file containing particle information, from which the central momentum will be set. The file contents are the same as required for elegant input with the awe_beam namelist.

track

- type: action command.
- function: track particles.

```
&track
    long center_on_orbit = 0;
    long center_momentum_also = 1;
&end
```

- `center_on_orbit` — A flag indicating whether to center the beam transverse coordinates on the closed orbit before tracking.
- `center_momentum_also` — A flag indicating whether to center the momentum coordinate also.

vary_element

- type: setup command.
- function: define an index and/or tie a parameter of an element to it.

```
&vary_element
    long index_number = 0;
    long index_limit = 0;
    STRING name = NULL;
    STRING item = NULL;
    double initial = 0;
    double final = 0;
&end
```

- `index_number` — A non-negative integer giving the number of the index.
- `index_limit` — A positive integer giving the number of values the index will take.
- `name` — The name of an element.
- `item` — The parameter of the element to tie to the index.
- `initial`, `final` — The initial and final values of the parameter.

3 Accelerator and Element Description

As mentioned in the introduction, `elegant` uses a variant of the MAD input format for describing accelerators. With some exceptions, the accelerator description for one program can be read by the other with no modification. Among the differences:

- `elegant` does not support the use of equations to compute the value of a quantity. The `link_element` namelist command can be used for this purpose, and is actually more flexible than the method used by MAD.
- `elegant` does not support substitution of parameters in beamline definitions.
- `elegant` contains many elements that MAD does not have, such as kick elements and numerically integrated elements.
- The length of an input line is not limited to 80 characters in `elegant`, as it is in MAD. However, for compatibility, any lattice created by `elegant` will conform to this limit.

`elegant`'s `print_dictionary` command allows the user to obtain a list of names and short descriptions of all accelerator elements recognized by the program, along with the names, units, types, and default values of all parameters of each element. At present, this serves as the only documentation of this information. The reader is referred to the MAD manual[2] for details on sign conventions for angles, focusing strength, and so forth.

4 Examples

Example runs and post-processing files are included along with the distribution of elegant. These are drawn from the author's research and all concern various aspects of the Argonne Positron Accumulator Ring (PAR) and its injection and ejection lines (LTP and PTB, respectively).

The examples are intended to demonstrate program capabilities with minimal work on the user's part. Each demo is invoked using a command (a C-shell script) that can both run elegant and post-process the output. After running the demo, the output can be viewed again without rerunning elegant by invoking the command with the word review added to the command line. Including the word hardcopy on the command line results in the graphs being sent to your default printer, which is assumed to accept Postscript.

The post-processing is typically handled by a lower-level script that is called from the demo script. These lower-level scripts are good models for the creation of customized scripts for user applications.

1. par10h* — These files provide a demonstration of Twiss parameter computation, tracking, element variation, and map analysis. The lattice is defined with kick elements, which are used for all tracking. After computation of the Twiss parameters for the PAR[6], a series of particles are tracked with different initial x coordinates. Finally, the tunes and Twiss parameters are computed by tracking; they are very close to the analytical values. The post-processing commands make phase-space plots and plots of FFTs of the motion, showing that the motion becomes chaotic at the stability limit. To execute this demo, type the command par10h.

2. par_sympl* — These files provide a demonstration of the symplecticity of tracking with elegant kick elements. A single large-amplitude particle is tracked for 2^{14} turns. The invariant J_x is then computed and plotted as a function of turn number. To execute this demo, type the command par_sympl. The post-processing takes quite some time because of the very large number of points.

3. par_chrom* — These files provide a demonstration of computing chromaticity and other parameters as a function of momentum offset using map analysis. The lattice is the same as par10h.lte, except all of the elements are implemented using second-order matrices. Hence, the chromaticity from tracking should be nearly identical to the analytical results computed by the twiss_output command, which it is. To run this demonstration, enter par_chrom. The reader may wish to try this demo again using ksbend, csbend, or nibend elements in place of the sbend elements, and kquad (ksext) elements in place of the quad (sext) elements.

4. par_damp* — These files provide a demonstration of damping partition calculation using single turn tracking with synchrotron radiation. The expected value of the longitudinal damping partition for PAR is $J_\delta = 1.758$. The user may edit the lattice file, par_damp.lte, to invoke a different element for the dipole magnet. In particular, definitions for numerically integrated dipoles with extended fringe-fields are present. To execute this demo, type the command par_damp.

5. par_dynap* — These files provide a demonstration of dynamic aperture runs for a series of randomized machines. Also exhibited here are orbit, tune, and chromaticity correction. The post-processing commands make a plot of the dynamic apertures with the physical aperture superimposed. (The orbcorr_plots script can also be used to plot orbit correction information.) To execute this demo, type the command par_dynap. The lattice has been stripped

down so that only a few of the more significant multipoles are present. Also, fictitious extra sextupoles have been added to compensate the lack of second-order edge terms in the bending magnets (these would result in nonsymplectic tracking if included). Still, the running time is many hours.

6. `ejoptk*` — These files provide a demonstration of the optimization of a multi-turn ejection bump for PAR, using a time-dependent kicker waveform (formed from two cubic splines). After optimization, the lattice is tracked with a realistic beam distribution to verify good transmission and show the centroid position vs z over three turns. To execute this demo, type the command `ejoptk`.

7. `ltp_te*` — These files provide a demonstration of transport line simulation. The Linac-to-PAR transport line is simulated with errors and trajectory correction to predict the transmission losses and the steering error at the exit of the septum. The trajectory correction uses tracking of a beam distribution, which is slower than tracking the centroid, but which produces better results in the presence of the large momentum spread. The reader may wish to verify this by turning off this feature and running the simulation again. To execute this demo, type the command `ltp_te`. The running time for this demo is quite long.

A Use of awe

awe format is a space-efficient binary format appropriate for storing large amounts of data, and is the primary format in which elegant writes output. Each awe file starts with a header, which defines the contents of the file. A series of "dumps" follows the header; these dumps are essentially separate tables of data. Each dump consists of a series of "points," each of which is basically a row of the table. The header defines the names and units of the columns of the tables. In addition, it defines the names and units of "auxiliary variables," which are quantities associated with each table that may take different values from dump to dump. Each dump of an awe file has a string label associated with it and each point of the dump has an additional string label associated with it. awe allows the user to select the data to be output based on wildcard matching with these labels, as well as by "filtering" based on the values of the data elements in the current line of the table.

A.1 Output Format Options

The program awe allows the user to extract data from awe-format files and create four different types of output:

- mpl-format output of any two quantities from the table. The basic usage for this type of output is:

  ```
  awe <input_filename> <output_file_rootname>
      -first=<quantity_name>[=<alias>],... -second=<quantity_name>[=<alias>],...
      [-process_dump[=<first_processing_mode>,<second_processing_mode>]
      [-separate_dumps]
  ```

  The output file rootname is the first part of a filename that awe will use to create filenames of the form `rootname_firstname_secondname.out`. The `-first` and `-second` switches allow one to specify the names of the first and second quantities for the output files; if you give a list of first and second quantities, they are paired for output in the order given. A quantity may also be assigned an "alias", which is simply a new name that will be used to generate the filename and make the labels in the mpl file.

  The `-process_dump` option allows processing of an entire table to produce a single number for each column. The possible processing options are `no`, `average`, `rms`, `sum`, `standard_deviation`, `mean_absolute_deviation`, `median`, `minimum`, `maximum`, `largest`, `first`, `last`, and `count` .

  The `-separate_dumps` option causes awe to create a separate mpl file for every dump in the input file. In elegant output, each dump typically corresponds to a different simulation step (i.e., a different randomized machine and/or beam, or a different point on the grid being swept). If this option is given, the filenames are of the form
  `rootname_dumpnumber_firstname_secondname.out`.

- mpl-format output of one or more tabulated quantities as a function of an auxiliary variable, with optional processing of the quantity. The basic format for this option is:

  ```
  awe input_filename output_file_rootname
      -sequence[=auxiliary_name[=alias]] -quantity=quantity_name[=alias],...
      [-process_dump=<processing_mode>]
  ```

The filenames created for the output data are of the form `rootname_auxiliaryname_quantityname.out`. Note that the `-process_dump` syntax is slightly different, since only one processing mode is required.

- `column`-format output of any number of quantities from the table. This produces a human-readable file that can also be used with the `column` program (see below). The basic usage for this option is:

  ```
  awe input_filename -table=output_filename,quantity_name[,quantity_name...]
  ```

  At present, dump processing is not available with `column`-format output; it will be added in the future.

- `contour`-format binary output for use with the `contour` program.

For all of these output options except the last, the user may specify a `printf` format string to tell `awe` how the data should be printed. This is done using the `-format` option. The default for `mpl`-format is "`%13.8e %13.8e`", while the default for `column`-format is "`%13.8e `" for each individual element. As an example, suppose one were extracting three quantities—one desired in floating-point format to two decimal places and the others in exponential format to six places; this would be accomplished by including

```
-format=''%.2f %.6e %.6e ''
```

on the command line. Note that the extra space at the end of the string is necessary to separate the last number from the label of the data point.

A.2 Selection of Data

Selection of data may be done using the `-dumps`, `-choose_dump`, `-filter`, and `-match_label` options. Of these, the first two allow selection of which dumps to use, while the last two allow selection of which points to use from within each dump.

- `-dumps` — Specifies that only dumps with certain "dump numbers" are to be included in the output. The dump number for an `awe` file starts at zero and increments by one for each dump. The syntax for this option is

  ```
  -dumps={<number> | (<begin>,<end>,<interval>) | <begin>,'...',<end> }, ...
  ```

  For example, if one wanted dump 1, all dumps from 10 to 20, and every other dump from 20 to 40, one would give

  ```
  -dumps=1,10,...,20,(22,40,2)
  ```

- `-choose_dump` — Specifies that only those dumps with labels matching a given string are to be included in the output. The string may include UNIX-style wildcards.

- `-filter` — Specifies that only those points for which the named quantity lies between the two limits given are to be included in the output.

- `-match_label` — Specifies that only those points with labels matching a given string are to be included in the output. The string may include UNIX-style wildcards.

A.3 Other Options

In this section are listed other options that can be used with `awe`:

- `-list_quantities[={'yes' | 'no' | 'only'}]` — This causes a list of the table quantities and auxiliary quantities to be printed to the screen. If `only` is given, the program exits immediately after doing this.
- `-verbose` — If given, this option causes `awe` to print information about each dump as it is processed.
- `-sort=<quantity_to_sort_by>[,<number_to_keep>[,'descending']]` — This causes each dump to be sorted prior to any processing or output. If `number_to_keep` is positive, only the first `number_to_keep` data points are retained after the sort.
- `-sample_interval=<n>` — This option causes `awe` to use only every n^{th} data point from each table.
- `-define_quantity=<quantity-name>,<quantity_unit>,<rpn-expression>` — This option defines a new quantity, to be added to each row of each table, in terms of existing or previously defined quantities. The definition is in terms of a `rpn` expression for the value of the new quantity. The value of any quantity for the current table and row is available by giving the name of the quantity. Similarly, the value of any auxiliary varirable for the current table is available by giving the auxiliary variable name.
- `-rpn_expression=<rpn-expression>` — This option passes a string directly to the `rpn` calculator subprogram. It is typically used to store values into memory for use in definitions.
- `-rpn_defns_file=filename` — This option allows the user to specify the name of a file containing `rpn` statements that is to be read in place of the default `rpn` definitions file. By default, `awe` reads the file given by the environment variable `RPN_DEFNS`.

B The mpl Scientific Toolkit

The mpl Scientific Toolkit is a group of programs, written in the C programming language[1], sharing a common data file format. The Toolkit is a powerful aid to a scientist's use of computers, alleviating much of the tedium associated with the analysis and interpretation of data.

In order to enhance the productivity of the Toolkit, the programs not only use mpl format (described below) for their input data, but they also create new data sets in the same format, when it is at all meaningful to do so. Hence, one can perform a sequence of operations on one's data by invoking a sequence of Toolkit programs.

The user interface for the programs is command-line-based, rather than menu-based or query-based. That is, the programs in the Toolkit are executed by typing the program name followed by a list of command-line options. This makes the programs more amenable to use in batch files and allows argument substitution through the command language.

C mpl Data Set Format

An mpl data set is an ordinary text file, such as might be created using a text editor. The lines of the file are read and processed one at a time, and no line may be longer than 1024 characters. The first four lines are descriptive text (as opposed to numeric data). The first line should be the name of the abscissa (or "x" variable), while the second line should be the name of the ordinate (or "y" variable). The third and fourth lines will be placed at the bottom and top of the plot, respectively, when the data set is plotted with mpl. The conventional format for the first and second lines is `<quantity-name> (<quantity_unit>)`; while this is not required, some programs detect this format in order to be able to alter the units automatically.

The fifth line contains the number of data points in the data set. It should be equal to or larger than the number actually in the data set. A warning will appear if this number is different from the actual number of points. (The program fixcount is available to count the points for you and put the right number on the fifth line of the data file.)

Every subsequent line contains one data point, each of which consists of two through four numbers. The first two values are x and y, respectively. If there are three values, the third is interpreted as the uncertainty (or sigma) for the y value. If there are four values, the third is interpreted as the x uncertainty, and the fourth as the y uncertainty. The number of values that are taken from each line is established by the number of values given for the first data point. Thus, if the first data point contains only two values, then only the first two values of each subsequent data point are used. Similarly, if the first data point contains four values, then all subsequent data points are expected to contain four values.

The data set ends when the file ends, or when the number of points specified has been reached, whichever comes first.

The format of the data is very free. Any non-numeric ASCII character (i.e., any character other than a digit, a plus or minus sign, or a period) can be used as a separator between data elements on a single line. Spaces and tabs are preferred, but solely for aesthetic reasons. Any line beginning with an exclamation point is treated as a comment, and ignored.

C.1 Command-Line Format of the mpl Programs

The next section contains short descriptions of the programs in the mpl toolkit at this time. A few of these programs are for very specific applications, though most are of general application. Some are quite simple, a few are downright trivial, while others are very sophisticated.

This list does not constitute a user's manual for these programs. Rather, the programs are intended to be self-documenting, since each program will respond with help information if it is run improperly. Hence, the user may obtain help for any program simply by running the program without any command-line input. The general form of the "usage message" that each program responds with is

```
usage: program_name required_argument1 required_argument2 ...
       [optional_argument_1] [optional_argument_2] ...
       {choice1.1 | choice1.2 | ...}
       {choice2.1 | choice2.2 | ...}
       [{optional_choice1.1 | optional_choice1.2 | ...}]
       [{optional_choice2.1 | optional_choice2.2 | ...}] .
```

In words: required arguments (e.g., the name of the input data set) are listed without delimiters. Optional arguments (usually a program-control "switch", or an optional output filename), are delimited by square brackets. Sets of arguments that the user *must* choose one and only one of are grouped by curly braces and separated by vertical bars. Optional sets of arguments that the user may choose one (and only one) of are grouped by curly braces inside square brackets.

Options, or "switches", are of the general form

```
-keyword[=value1[,value2...]],
```

with several alternative forms recognized: you may use / instead of -, and : or , instead of =. Any keyword may be abbreviated when enter on the command line, so long as enough characters are supplied to make the keyword uniquely identifiable. If the value listed in the usage message has single quotes around it, then the value must be typed literally (i.e., as in `-average=rms`); you should not type the single quotes yourself in running the program. Such values may, like the keywords, be abbreviated. In contrast, a value listed in double quotes represents a string that *may* need to be enclosed double quotes (if, for example, it contains spaces, as in `-title=''x vs y''`).

In what follows, the phrase "data set" refers to data stored in an `mpl` format file, which consists of a set of data points (x_n, y_n) with optional uncertainties σ_{y_n} and σ_{x_n}. Unless otherwise stated, each program writes its results as a new data set in a file of the user's choice. Most programs accept the `-format` argument to allow the user to select the output format of the data; refer to the previous section for a discussion.

C.2 Descriptions of the `mpl` Programs

`add`: Adds (or subtracts) two data sets, with error propagation. Normally, the ordinates are added assuming that the abscissae match. Optionally, the abscissa can be added. There is also an option to match the abscissae before adding the ordinates, or vice-versa.

`column`: Extracts data from generic tables, such as might be output by any number of programs, and creates data sets. Hence, a program that prints rows of quantities in many columns can be interfaced to the Toolkit via `column`. (A better way to accomplish this is to modify the program to write its data in either `mpl` or `awe` format.) Also, `column` extracts data from `column`-format files into `mpl`-format files.

`combine`: Combines any number of data sets into a new data set, sorting the data sets by the first abscissa value in each set.

`convol`: Performs discrete Fourier convolution or deconvolution of two data sets, assuming that the range and number of points is the same for both sets.

`deriv`: Takes the derivative of a data set, using

$$\left(\frac{\partial F}{\partial x}\right)_{x=x_a} \approx \frac{F(x_{n+m}) - F(x_{n-m})}{x_{n+m} - x_{n-m}}, \tag{1}$$

where $x_a = (x_{n+m} + x_{n-m})/2$ and where m is specified by the user. Strictly, the points should be equispaced (if not, use m=1).

`dsc`: The Data Set Calculator is one of the most powerful programs in the Toolkit, being something like a command-line spreadsheet. It allows use of an arbitrary number of input data files to create an arbitrary number of output data files using user-specified equations for the new data.

`envelope`: Processes any number of input files, assumed to have identical numbers of points and identical abscissa values, producing optional output files containing the minima, maxima, averages, mean-absolute-deviations, standard-deviations, RMS values, and/or sums of ordinate values at each abscissa.

`filter`: Performs digital filtering of a data set by doing an FFT, applying the filter, then doing an inverse FFT. Supports low-pass and high-pass filters with linear roll-off, as well as Parzen windowing[7] of the data.

`fiteval`: Assumes that one data set is a fit to the other, and evaluates how good a fit it is.

`fixcount`: Fixes the point count of data sets. `fixcount` will attempt to simply overwrite the existing point count, if the fifth line of the data set has a sufficient number of characters to allow this without running over onto the next line stored in the file. E.g., if the fifth line contains `''1<return>''` and the actual point count is 100, fixcount will have to re-write the entire file to make room for the two zeroes. If, however, the fifth line has a number of trailing spaces, all `fixcount` need do is replace some of the trailing spaces with zeroes. This is not a trivial concern for data sets with thousands of points, and hence it is a good idea to put trailing spaces on the fifth line of each data set.

`fft`: Performs the FFT of a data set, producing a new data set with the magnitude vs. the frequency. Also does optional windowing[7] and provides output of real and imaginary components. Will work in either single or double precision, and will optionally pad or truncate your data to achieve 2^n data points (necessary for the FFT).

`fwhm`: Computes the full-width at half-maximum, with optional smoothing.

`gfit`: Fits a gaussian to a data set, finding the sigma, mean, baseline, and height, with the option of generating a new data set containing the fit evaluated at a series of points.

`ggen`: Generates a data set from evaluation of a gaussian at equispaced points.

`hist`: Makes histograms and cumulative distributions of one or more data sets, with optionally filtering and weighting of the histogrammed variable by the other variable. Optionally does various normalizations of the histogram and does statistical analysis of the data.

`integ`: Computes the integral of a data set using the trapizoid rule with error propagation, creating a new data set with the integral as a function of the abscissa.

`interp`: Does polynomial interpolation on a data set, using whatever order of polynomial the user requests, with options for creating a new data set with interpolated values at equispaced points and for transforming one variable of a data set via interpolation on the first data set.

`lsf`: Does error-weighted least-squares fits to any order using either ordinary polynomials or Chebyshev T polynomials. Options for fitting only even or odd polynomials, for fitting only specified orders, and for automatic elimination of orders that (though requested) are not genuinely present. Also provides difference data sets and data sets from evaluation of fitted polynomials.

`mpl`: Versatile plotting program for multiple data sets. Allows zooming, point plotting, symbol plotting, and much more. Supports many common graphic output devices through use of drivers from GNUPLOT. Has Greek and scientific character sets, with subscripting, superscripting, and in-line control of character attributes. The following "escape sequences" are recognized in character strings:

1. \$g, \$r : switch to Greek or Roman characters.
2. \$a, \$b, \$n: go to Above (superscript), Below (subscript), or Normal script.
3. \$s, \$e: Start and End special (mathematical) symbols.
4. \$i, \$d: Increase ($\times 1.5$) or Decrease ($/1.5$) character size.
5. \$t, \$f: go to Taller or Fatter letters (these are inverses of each other).
6. \$u, \$v: displace text vertically Upward or downward (respectively) by one-half character-height.
7. \$h: back-space one-half character width.

`mult`: Does pair-wise multiplication (or division) of the ordinates (or abscissae, or both) of two data sets, with error propagation. If the data sets are of unequal length, the program will optionally try to line up the abscissae before multiplying.

`murge`: Merges two data sets, in the sense of taking the abscissae and ordinates of the new data set from the abscissae and ordinates of two other data sets, as specified by the user. (The strange spelling is to avoid confusion with the VAX/VMS MERGE utility.)

`params`: Computes many parameters of a series of data sets, including positions, heights and sharpnesses of peaks, positions and values of minima and maxima, averages, and medians.

`peakfind`: Finds peaks in a data set, with optional smoothing and a user-defined threshold.

`qsort`: Sorts a data set into ascending (or descending) order by the abscissa, subsorting by the ordinate, with optional elimination of duplicate points.

`relabel`: Alters the labels in an `mpl` file, composing new labels from the old labels, from those in another file, or using new labels supplied on the command line.

`rescale`: Arguably the most powerful program in the toolkit. Performs very versatile transformations of a data set, including normalization, centering, scaling and offsetting, taking the logarithm, and more. Data sets may be sparsed, windowed, reordered, and the abscissae and ordinates may be swapped. Propagation of errors can be performed if sigmas are given in the data set. `rescale` will also accept user-defined transformations, specified as `rpn` equations.

`rndgen`: Generates random-number pairs, with gaussian and uniform distributions supported. The random number generator uses a random shuffling routine with two linear congruential generators.

`smooth`: Smooths a data set by multipass averaging over adjacent points.

`setlog`: Performs set logic on two data sets. For example, `setlog` will find all the data points in one data set but not the other, or all the data points in both sets. User-specified tolerances are accepted to determine when two points are "the same."

`stats`: Does statistical analyses of a data set, computing various moments, widths, and cumulative distribution parameters. Also gives the minimum, maximum, and spread.

`tellipse`: Creates a data set containing points along an ellipse defined in terms of the Twiss parameters and emittance.

`total`: Creates a new data set with

$$\tilde{y}_n = \sum_{i=1}^{n} y_i \tag{2}$$

`zerofind`: Finds locations of zeros in a data set, with interpolation between points.

D The rpn Calculator

The program rpn is a Reverse Polish Notation programmable scientific calculator written in C. It is incorporated as a subprogram into `elegant`, `awe`, and a number of the `mpl` programs. It also exists as a command-line program, `rpnl`, which executes its command-line arguments as rpn operations and prints the result before exiting. Use of rpn in any of these modes is extremely straightforward. Use of the program in its stand-alone form is the best way to gain familiarity with it. Once you've entered rpn, entering "help" will produce a list of the available operators with brief summaries of their function. Also, the rpn definitions file `rpn.defns`, distributed with `elegant`, gives examples of most rpn operation types.

Like all RPN calculators, rpn uses stacks. In particular, it has a numeric stack, a logical stack, and a string stack. Items are pushed onto the numeric stack whenever a number-token is entered, or whenever an operation concludes that has a number as its result; items are popped from this stack by operations that require numeric arguments. Items are pushed onto the logical stack whenever a logical expression is evaluated; they are popped from this stack by use of logical operations that require logical arguments (e.g., logical ANDing), or by conditional branch instructions. Items enclosed in double quotes are pushed onto the string stack; items are popped from this stack by use of operations that require string arguments (e.g., formatted printing).

rpn supports user-defined memories and functions. To create a user-defined memory, one simply stores a value into the name, as in "1 sto unity"; the memory is created automatically when rpn detects that it does not already exist. To create a user-defined function, enter the "udf" command; rpn will prompt you for the function name and the text that forms the function body. To invoke a UDF, simply type the name.

A file containing rpn commands can be executed by pushing the filename onto the string stack and invoking the "@" operator. rpn supports more general file I/O through the use of functions that mimic the standard C I/O routines. Files are identified by integer unit numbers, with units 0 and 1 being permanently assigned to the terminal input and terminal output, respectively.

References

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., second edition, 1988.

[2] H. Grote, F. C. Iselin, "The MAD Program–Version 8.1," CERN/SL/90-13(AP), June 1991.

[3] K. L. Brown, R. V. Servranckx, "First- and Second-Order Charged Particle Optics," SLAC-PUB-3381, July 1984.

[4] M. Borland, "A High-Brightness Thermionic Microwave Electron Gun," SLAC-Report-402, February 1991, Stanford University Ph.D. Thesis.

[5] H. A. Enge, "Achromatic Mirror for Ion Beams," Rev. Sci. Inst., 34(4), 1963.

[6] M. Borland, private communication.

[7] W. H. Press, *et al*, *Numerical Recipes in C*, Cambridge University Press, Cambridge, 1988.